Radiology: Artificial Intelligence

EDITORIAL

Magician's Corner: 6. TensorFlow and TensorBoard

David C. Vogelsang, BS • *Bradley J. Erickson, MD, PhD*

From the Department of Radiology, Mayo Clinic, 200 First St SW, Rochester, MN 55905. Received February 2, 2020; revision requested March 4; revision received April 9; accepted April 16. **Address correspondence to** B.J.E. (e-mail: *bje@mayo.edu*).

Conflicts of interest are listed at the end of this article.

Radiology: Artificial Intelligence 2020; 2(3):e200012 • https://doi.org/10.1148/ryai.2020200012 • Content code: IN

In a previous article, we provided an introduction to TensorFlow and used it to build a U-Net for the purpose of image segmentation (1). TensorFlow is a popular framework for deep learning applications, developed by Google and first released in 2015. TensorFlow version 2 was released in late 2019 and has several important design changes that make it much more approachable for those new to deep learning, such as direct integration with Keras, while also supporting features for advanced users. The TensorFlow name comes from the fact that tensors are the fundamental computational objects in deep learning: A tensor is an n-dimensional array. Computations on those tensors are expressed as stateful dataflow graphs. TensorFlow enjoys broad hardware and software support, with versions available for tiny microcontrollers up to some of the most powerful computers in the world.

A Classifier in TensorFlow

In the first article of this series, we used FastAI (which is based on PyTorch) to build a classifier based on a ResNet-34 model to classify medical images into one of six classes (CT Head, CT Chest, CT Abdomen, Chest Radiograph, MR Brain, or MR Breast) (2). This classifier required only a few lines of code using the defaults provided by FastAI.

In this article, we will use TensorFlow to build a simpler classifier from scratch. This classifier will allow us to experiment with designs where the network is too simple (resulting in underfitting) or too complex (resulting in overfitting). We will use a DataGenerator to feed augmented data to our network. Finally, we will use TensorBoard to monitor training progress to recognize when problems like overfitting are occurring.

The first step is to open colab.research.google.com and retrieve the notebook for this article (open a file, use the Github tab, search for "RSNA," and select MC6_TensorFlow_and_TensorBoard.ipynb). Once you have opened this notebook, run cell 1, which loads in our libraries. Note that Google has updated its Colab notebooks to use TensorFlow version 2 by default. Cells 2 and 3 load and rearrange data to make it more compatible with TensorFlow. Please run cells 1, 2, and 3 now.

Cell 4 introduces an important tool for training on large datasets, DataGenerators. Having a large number of training examples is critical to achieve top performance. With large datasets, we often need to split training sets into "batches" that are loaded into the memory of the GPU one at a time, since the entire training set often cannot fit into the GPU memory. The same can be true of the host computer's memory; very large training sets, particularly if one is augmenting data, may not fit into the random access memory (RAM) of the host, and so for this reason, we use generators. A generator is a computer science construct that allows one to operate sequentially on many examples (usually coming from disk storage in the case of machine learning) without having to have all examples stored in RAM. With this technique, it is not necessary to load all the images into memory simultaneously; we can load one example at a time, create the augmented versions of the example, feed the augmented versions into the neural network, and complete the cycle by removing the augmented versions from RAM. The statement on line 8 of cell 4 creates the DataGenerator and describes the ways the image will be augmented. Line 16 connects this DataGenerator with a data source, which in our case is a directory of training images. (Note from the name that these images are for creating the training examples.) The next few lines create a second DataGenerator for our validation set. Run cell 4 to create these data generators.

In cell 5, we "wrap" our DataGenerator as a tf.data.Dataset. The tf.data application programming interface (API) enables one to build complex input pipelines from simple, reusable pieces (for more information, see *https://www.TensorFlow.org/guide/data*). Unfortunately, the tf.data API doesn't have all the functions we want for data augmentation, such as the Keras ImageDataGenerator provides. To access that, we need to wrap the DataGenerator as a tf.data.Dataset. Run cell 5.

In cell 6, we create our classifier, a simple convolutional neural network. Line 4 of the cell creates a sequential computational graph, the typical form for deep learning. The next lines describe the convolution and pooling layers for our network; we will use three of each. Note here that you specify the size of the convolutional kernel, the number of filters, and the activation function. Padding refers to how to handle the edges of the images: in nearly all cases you will want 'same' to keep the convolved image size the same as the input image size. After the convolution and pooling layers, we flatten the coefficients into a one-dimensional array (line 12), followed by a fully connected layer ('Dense' in TensorFlow parlance). The Dropout parameter is the fraction of nodes to be randomly removed; values of 30%–50% (0.3–0.5) are typical. In line 21, we compile the network ('Computational Graph' in TensorFlow parlance), specify the popular adaptive optimizer Adam (3), and use categorical_crossentropy (categorical since these are classes, and cross-entropy because that works well for measuring the difference between the various class predictions). Run cell 6.

Visualizing Metrics

In cell 7, we train our model. In line 8, we set up early stopping to define conditions under which training will stop before executing the specified number of epochs. For instance, if we do not see an improvement in the validation set loss for a few iterations, we have probably maximized our training, and more epochs will result in overfitting. In line 6, we create a variable, log_dir, which is a folder name where performance data are written as we train the network. Adding a timestamp to the directory name can be useful because it ensures you don't overwrite any of your prior logs. We could add other descriptors to the name, but we will just use the timestamp for now. Next, using tf.keras.callbacks.TensorBoard, we create a callback that logs our metrics to log_dir every epoch. Finally, we fit our model calling model.fit (). The parameters are self-explanatory and include pointers to the training and validation data and to the callback(s) we wish to use during training. Run cell 7.

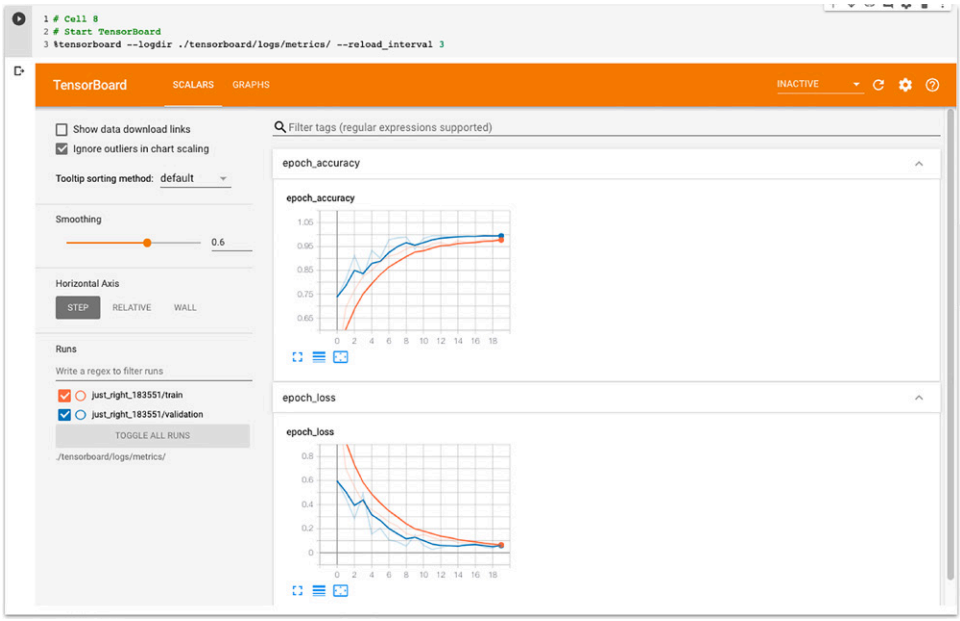

Figure 1: The TensorBoard user interface. There are two tabs at the top: Scalars and Graphs. Scalars will graph scalar data such as the loss and accuracy values that are written to the logging directory. You can force the interface to read through the logging directory and redraw by clicking the update circular arrow in the upper right and also adjust the preferences by clicking the gear icon. On the left, you can select what is displayed and how it is displayed. By default, the user interface will perform some smoothing and display all data.

In cell 8, we can launch TensorBoard from our notebook. Let's take a look at each of the tabs on the top navigation bar of TensorBoard (Fig 1). The Scalars dashboard shows how the loss and metrics change with every epoch. You can use it to track training speed, learning rate, and other values. The Graphs dashboard displays the model. The Distributions and Histograms dashboards show the distribution of a tensor over time. This information can be useful to visualize weights and biases and verify that they are changing in an expected way. More detailed information is available at *https://www.tensorflow.org/tensorboard/get_started*.

Colab notebooks are a viable way to train networks, but most people train on their local computer. In that case (or with a local server), it is more typical to launch TensorBoard before starting training and to point it to the logging directory where TensorFlow will write results as training proceeds. In this case, you can observe the loss curves as the network is training. In our case, Colab only lets one cell execute at a time, which means that TensorBoard displays information after training.

Visualizing Overfitting

Using TensorBoard to visualize our training metrics, such as training and validation loss and accuracy, can help us recognize if we are overfitting, underfitting, or just right. In this next section, we will be demonstrating overfitting. Overfitting is the point in training when we have gone from learning to memorizing. At this point, our model will no longer generalize well. If you're watching TensorBoard, it is the point where training loss continues to head toward zero (improve), but validation loss starts to increase dramatically (get worse). We have previously described ways to prevent overfitting: *(a)* add more training data, possibly by using data augmentation; *(b)* reduce model complexity (fewer layers, fewer nodes, fewer filters, etc); *(c)* add dropout (which also reduces nodes but in a random way); and *(d)* add regularization (L1 or L2).

To demonstrate overfitting, we reduce the size of our training dataset and get rid of data augmentation (cell 9) and create a model that is more complex than needed with many layers and many nodes in each layer (cell 10). After training the model in cell 11, you can see how badly our model overfit in TensorBoard by running cell 12. You will see the validation loss begins to increase (lower accuracy), while the training loss continues to decrease (accuracy keeps getting better) (Fig 2). This divergence indicates that the network is learning the training examples and is not generalizing well to the validation set. Note also that we unchecked the just_right logs so that only the overfitting curves are shown.

Underfitting

Underfitting occurs when there is not enough power in the model to learn the problem: It can be recognized as low performance on both training and validation sets. In cell 13 is a very simple, low-power model. Run cells 13, 14, and 15 to create the very simple model, fit, and then display the result in TensorBoard. As an exercise, add layers to the model until it performs reasonably well, without overfitting. (If you get too many old log files that clutter up the graphs, you can delete the prior log files by uncommenting line 3 of cell 15.)

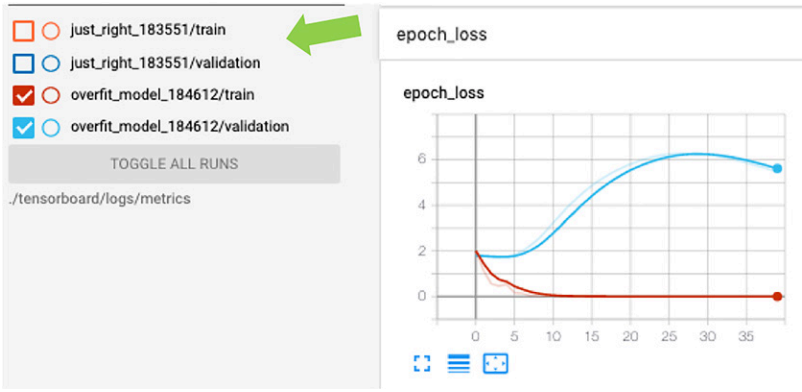

Figure 2: The lower part of the TensorBoard display is where to uncheck the 'just_right' logs, so that only the overfit are shown (green arrow). The characteristic sign of overfitting is seen here, where the validation set loss starts to go up, even as the training set loss continues to drop (improve). It is very important to review your training and validation set loss curves to help identify overfitting.

Visualizing Training Images

There are times when it is useful to review the images that are used for training. The TensorFlow Image Summary API allows us to view tensors as images in TensorBoard. This feature is helpful when using data augmentation to ensure that the augmented examples are useful for training; we can sample our training data augmentations and see if they make sense or not. We can also review our dataset to look for mislabeled data.

In cell 16, we use tf.summary.image to display a batch of training images from our dataset that have been augmented. First, we create a string, logdir_train_data, that contains a path and folder name to which to write our data. Next, we create file_writer to write our tensors to a log we can open in TensorBoard. The call to next (iter (train_ds)) in line 12 retrieves a batch of tensors from our tf.data.Dataset dataset (which was obtained from the Keras ImageDataGenerator). Finally, the tensors are written to the logdir_train_data folder, and TensorBoard can then access them and displays them as images.

Plotting a Confusion Matrix

A confusion matrix can help visualize the performance of our classification model. We can see which images the model commonly misclassifies, or confuses, for other images. To create a confusion matrix in TensorBoard, one requires some functions to organize the data in a suitable fashion and then create the graphical image of the matrix (cell 17). The default approach (*https://www.TensorFlow.org/tensorboard/image_summaries*) has been modified to work with our model. We must then retrain the network, with calls to these functions as training proceeds. Important note: when you compile the model, the loss function needs to be set to 'categorical_crossentropy.' Also, if you altered class_mode in cell 4 with the DataGenerator and set class_mode to 'sparse_categorical_crossentropy,' you will need to change it back to 'categorical_crossentropy.' Run cells 17 and 18 now. In cell 19, we launch TensorBoard to display the confusion matrix. Confusion matrices are useful to understand which classes are resulting in the most errors. In some cases, you can alter the preprocessing of the images or take other steps to improve classification performance.

Conclusion

Deep learning technology is extremely powerful. However, it is important to have tools to monitor these algorithms as they are working. Although the examples in these articles are specifically tuned to give good performance in just minutes, real-world cases of deep learning training may require hours to days of training. Tools such as TensorBoard provide a way to monitor training to avoid wasting hours and days of computation. These tools also can provide a way to detect overfitting and underfitting and to see the problems or classes that are degrading performance.

Author contributions: Guarantors of integrity of entire study, D.C.V., B.J.E.; study concepts/study design or data acquisition or data analysis/interpretation, D.C.V., B.J.E.; manuscript drafting or manuscript revision for important intellectual content, D.C.V., B.J.E.; approval of final version of submitted manuscript, D.C.V., B.J.E.; agrees to ensure any questions related to the work are appropriately resolved, D.C.V., B.J.E.; literature research, D.C.V., B.J.E.; experimental studies, D.C.V.; and manuscript editing, D.C.V., B.J.E.

Disclosures of Conflicts of Interest: **D.C.V.** disclosed no relevant relationships. **B.J.E.** Activities related to the present article: disclosed no relevant relationships. Activities not related to the present article: on the board of FlowSIGMA, which does workflow for AI, but neither author nor institution are paid. Other relationships: disclosed no relevant relationships.

References

1. Erickson BJ, Cai J. Magician's corner: 4. image segmentation with U-Net. Radiol Artif Intell 2020;2(1):e190161.
2. Erickson BJ. Magician's corner: how to start learning machine learning. Radiol Artif Intell 2019;1(4):e190072.
3. Kingma DP, Ba J. Adam: a method for stochastic optimization. ArXiv [cs.LG]. [preprint] http://arxiv.org/abs/1412.6980. Posted 2014. Accessed October 12, 2018.